R Tutorial

S. P. Blomberg

This self-tutorial is a short introduction to the R statistical software package for those who have had no previous exposure to it. It may largely be a refresher if you have had any previous R experience.

Obtaining R

R can be obtained from the Comprehensive R Archive Network, or CRAN `https://cran.r-project.org/`. RStudio, an interactive development environment (IDE) for R can be found at `https://www.rstudio.com/`. Use of RStudio as a front-end to R is recommended because it will help you organise your R session and RStudio has many features that make using R much easier.

After installing R and RStudio, start RStudio in the usual way. R will open automatically within RStudio. You are ready to begin. The following are some simple commands and associated output which will help you to gain a basic understanding of the S language, which is the language that R uses. Everything on a line after a hash (`#`) is a comment (which you don't need to type). Further, output from R is preceded by a `##` (You won't see this in your output. It is just for convenience in this document.), and often by a "`[1]`". The command line prompt for R is "`>`".

After doing the tutorial you should experiment with your own examples and make further use of the "`help`" function. More help can be obtained by typing `help.start()`, which should launch a web browser interface to all the help files. This tutorial is best used by printing it out and typing in the commands yourself. Please don't copy and paste into R. The idea is to get you used to interacting with R *via* RStudio and the command line interface.

Basic Calculations

```
> 3 + 5   # Find the sum of 3 and 5 (press Enter after typing the 5)

## [1] 8

> 5 - 2   # Subtract 2 from 5

## [1] 3

> 5 * 3

## [1] 15

> 6 * (7 - 3)

## [1] 24

> 12 / 3

## [1] 4

> 3^2

## [1] 9

> 3^(1/2)         # Calculate the square root of 3

## [1] 1.732051
```

Display Options

```
> print(3^(1/2), digits = 10)     # Display the square root of 3 to 10 significant digits

## [1] 1.732050808

> options(digits = 4)     # Set the number of significant digits to 4 from now on
> 3^(1/2)

## [1] 1.732

> options(digits = 7)     # Reset the number of significant digits to 7
> 3^(1/2)

## [1] 1.732051
```

Basic Functions

```
> sqrt(3) # Calculate the square root of 3 using function sqrt()

## [1] 1.732051

> help(sqrt) # Get help for the sqrt function
> ?sqrt   # Shortcut for help
> exp(1)  # Display the transcendental number e

## [1] 2.718282

> log(2.718)      # Find the natural logarithm of 2.718

## [1] 0.9998963

> log10(100)

## [1] 2

> logb(8, base = 2)

## [1] 3

> pi      # Display the ratio of a circle's circumference to its diameter

## [1] 3.141593

> sin(pi/2) # Calculate the sine of pi/2 in radians

## [1] 1

> help(sin)
> asin(1) # Calculate the arcsine (inverse sine) of 1

## [1] 1.570796
```

Scalars

```
> a <- 3          # Create a scalar object with the name a and value 3
> a               # Display the value of a

## [1] 3
```

```
> a^2              # Find the square of a

## [1] 9

> a^4

## [1] 81

> b <- a^2         # Create another object b with the value a^2 (ie 9)
> b                # Display the value of b

## [1] 9

> a <- 10          # Change a so it equals 10 (no longer 3)
> A <- 11          # Create an object A equal to 11 (NB A is not the same as a)
> const <- (a + A) * a     # Create another constant
> const

## [1] 210
```

Listing and Removing Objects

```
> ls()    # List all the variables so far defined

##  [1] "a"           "A"           "arr"         "b"           "const"
##  [6] "dat"         "FUNNYFUNC"   "i"           "j"           "mat"
## [11] "p1"          "p2"          "PRODANDSUM"  "QUAD.ROOTS"  "res"
## [16] "sampn"       "sampndat"    "sampu"       "x"           "y"
## [21] "z"

> rm(a, b)         # Remove the objects a and b
> ls()    # Display the objects which now exist

##  [1] "A"           "arr"         "const"       "dat"         "FUNNYFUNC"
##  [6] "i"           "j"           "mat"         "p1"          "p2"
## [11] "PRODANDSUM"  "QUAD.ROOTS"  "res"         "sampn"       "sampndat"
## [16] "sampu"       "x"           "y"           "z"

> rm(list = ls()) # Delete all objects in existence
> ls()    # Check that everything has been deleted

## character(0)

> A       # Try to display the object A (which now doesn't exist)

## Error in eval(expr, envir, enclos):  object 'A' not found
```

Vectors

```
> y <- c(3, 2, 4, 0, 3)    # Create a column vector of length 5
> y       # Display the elements of y (they appear in a row)

## [1] 3 2 4 0 3

> t(y)    # Display the transpose of y (a row vector)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    3    2    4    0    3
```

```
> t(t(y)) # Display the elements of y (a column vector) in a column

##      [,1]
## [1,]    3
## [2,]    2
## [3,]    4
## [4,]    0
## [5,]    3

> y[3]    # Display the 3rd element of y

## [1] 4

> y[c(3, 5)]      # Display the 3rd and 5th elements of y

## [1] 4 3

> y[-3]   # Display y with the 3rd element removed

## [1] 3 2 0 3

> y[-c(3, 5)]     # Display y with the 3rd and 5th elements removed

## [1] 3 2 0

> sort(y) # Sorts the values of y in ascending order

## [1] 0 2 3 3 4

> sort(y, decreasing = TRUE)      # Sorts the values of y in descending order

## [1] 4 3 3 2 0

> 1:5     # The integers from 1 to 5

## [1] 1 2 3 4 5

> 5:1     # Display the integers from 5 to 1

## [1] 5 4 3 2 1

> rank(y) # Find the ranks of the values in y

## [1] 3.5 2.0 5.0 1.0 3.5

> unique(y)       # Display the unique values in y

## [1] 3 2 4 0

> y[y > 1]        # Display the values in y which are greater than 1

## [1] 3 2 4 3

> length(y[y > 1])        # Find the number of elements in y which are greater than 1

## [1] 4

> y[y >= 3]       # Display the elements in y equal to 3 or more

## [1] 3 4 3

> length(y[y == 3])       # Find the number of 3s in y

## [1] 2

> length(y[ y != 3])      # Find the number of elements in y that are not equal to 3
```

```
## [1] 3

> summary(y)      # Display some simple summary statistics for the values in y

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0     2.0     3.0     2.4     3.0     4.0

> summary(as.factor(y))   # Count each distinct value in y (eg. there are two 3s)

## 0 2 3 4
## 1 1 2 1

> y + 5   # Add 5 to each value in y

## [1] 8 7 9 5 8

> sum(y)  # Sum the values of y

## [1] 12

> help(sum)       # Get help on the function sum
> cumsum(y)       # Find the cumulative sums for the values in y

## [1]  3  5  9  9 12

> prod(y) # Find the product of all the values in y

## [1] 0

> y * y   # Find the squares of the values in y

## [1]  9  4 16  0  9

> sum(y * y)      # Sum the squares of y in another way

## [1] 38

> mean(y) # Sample mean

## [1] 2.4

> var(y)  # Sample variance

## [1] 2.3

> sd(y)   # Sample standard deviation

## [1] 1.516575

> sqrt((1/(length(y)-1))*sum((y-mean(y))^2)) # find the sd in a different way

## [1] 1.516575

> y[3] <- 100     # Change the 3rd element of y from 4 to 100
> y

## [1]   3   2 100   0   3

> y[3] <- 4 # Change the 3rd element of y back to 4
> y

## [1] 3 2 4 0 3
```

Matrices

```
> help(matrix)    # Get information on the matrix() function
> mat <- matrix(1:6, nrow=2, ncol=3) # Create a 2 by 3 matrix called mat
> mat     # Display the matrix mat

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

> mat[2, 3]       # Display the value of mat in the 2nd row and 3rd column

## [1] 6

> mat[2, ]        # Display the 2nd row of mat

## [1] 2 4 6

> mat[, 3]        # Display the 3rd column of mat

## [1] 5 6

> sum(mat)        # Sum all the values in mat

## [1] 21

> colSums(mat)    # Sum the values in each column of mat

## [1]  3  7 11

> rowSums(mat)    # Sum the values in each row of mat

## [1]  9 12

> apply(mat, 1, sum)      # Sum the rows another way

## [1]  9 12

> apply(mat, 2, sum)      # Sum the columns another way

## [1]  3  7 11

> help(apply)     # Get more information on apply()
> t(mat)  # Display the transpose of mat

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6

> mat %*% t(mat)  # Post-multiply mat by its transpose

##      [,1] [,2]
## [1,]   35   44
## [2,]   44   56

> t(mat) %*% mat  # Pre-multiply mat by its transpose

##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61

> z <- 1:4        # Create the vector (1, 2, 3, 4)
> sum(z^2)        # Find thes sum of squared values in z
```

```
## [1] 30

> t(z) %*% z       # Find the sum of the squared values in z using matrix multiplication

##      [,1]
## [1,]   30

> z %*% t(z)       # Post-multiply z by its transpose to create a 4 by 4 matrix

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
```

Arrays

```
> help(array)
> arr <- array(c(1:8, 11:18, 111:118), dim=c(2,4,3))      # Create a 2 by 4 by 3 array
> arr

## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   11   13   15   17
## [2,]   12   14   16   18
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]  111  113  115  117
## [2,]  112  114  116  118

> arr[1, 4, 3]     # value of the 3rd matrix of arr at the 1st row and 4th column

## [1] 117
```

Functions

Type out the following lines into the top left window of RStudio. Then select the code and press the “run” button. You will thereby create a new function called FUNNYFUNC. Do likewise for the subsequent examples.

```
FUNNYFUNC <- function (x, y) {
# This function returns the sum of x and y if x < y,
# and the product of x and y if x >= y
if (x < y) {res <- x + y} else {
        res <- x * y }
res
}
```

```
> FUNNYFUNC(3, 4) # Apply the function with x = 3 and y = 4

## [1] 7

> FUNNYFUNC(4, 3) # Apply the function with x = 4 and y = 3

## [1] 12
```

```
PRODANDSUM <- function (x, y) {
        # This function calculates the product and sum of two numbers
        # and returns a list with these two quantities as its components
        # Inputs: x = first number
        #   y = second number
        #
        # Outputs: product = product of x and y
        # sum = sum of x and y
        pr <- x * y
        su <- x + y
        list(product=pr, sum=su)
}
```

```
> res <- PRODANDSUM(2, 3) # Applyt the function with x=2 and y=3
>         # so as to create a new object called res
>         # which is a list with two components
>         # called product and sum
> res$product     # Display the first component of res

## [1] 6

> res$sum # Display the second component of res

## [1] 5

> res     # Display both components of res

## $product
## [1] 6
##
## $sum
## [1] 5

> res[[2]]        # Display the second component of res another way

## [1] 5
```

```
QUAD.ROOTS <- function (a, b, c) {
        # This function calculates the roots of the quadratic equation
        # a*x^2 + b*x + c, where a != 0 and b^2 - 4*a*c >= 0.
        # Inputs: a = coefficient of squared term
        # b = coefficient of linear term
        # c = constant term
        # Outputs: vector with the two roots (possibly the same),
        # or an error message if a = 0 or there are no real roots.
if (a == 0) stop("This equation is not quadratic since a == 0")
                d <- b^2 - 4*a*c
                if (d >= 0) res <- (-b+c(1,-1)*sqrt(d))/(2*a) else {
                        stop("This quadratic equation has no real roots.")
                        }
res
}
```

```
> QUAD.ROOTS(3, -4, 1)

## [1] 1.0000000 0.3333333

> QUAD.ROOTS(0, -4, 1)

## Error in QUAD.ROOTS(0, -4, 1):  This equation is not quadratic since a == 0

> QUAD.ROOTS(3, -4, 2)

## Error in QUAD.ROOTS(3, -4, 2):  This quadratic equation has no real roots.

> QUAD.ROOTS(1, -2, 1)

## [1] 1 1
```

Congratulations. You have written your first three functions in R. Most of the built-in functions in R are also written in the R language. You can usually see the definition of the function by simply typing its name at the R prompt (without the parentheses). Some functions have very short definitions, but others can be much longer. For example, the `sd()` function is very short (it just takes the square-root of the variance):

```
> sd

## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x564c7660fd70>
## <environment: namespace:stats>
```

But the `lm()` function, used to fit General Linear Models (regression, ANOVA, etc.) is quite long and detailed:

```
> lm

## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## {
##     ret.x <- x
##     ret.y <- y
##     cl <- match.call()
##     mf <- match.call(expand.dots = FALSE)
##     m <- match(c("formula", "data", "subset", "weights", "na.action",
##         "offset"), names(mf), 0L)
##     mf <- mf[c(1L, m)]
##     mf$drop.unused.levels <- TRUE
##     mf[[1L]] <- quote(stats::model.frame)
##     mf <- eval(mf, parent.frame())
##     if (method == "model.frame")
##         return(mf)
##     else if (method != "qr")
##         warning(gettextf("method = '%s' is not supported. Using 'qr'",
##             method), domain = NA)
##     mt <- attr(mf, "terms")
##     y <- model.response(mf, "numeric")
##     w <- as.vector(model.weights(mf))
##     if (!is.null(w) && !is.numeric(w))
##         stop("'weights' must be a numeric vector")
##     offset <- as.vector(model.offset(mf))
##     if (!is.null(offset)) {
##         if (length(offset) != NROW(y))
##             stop(gettextf("number of offsets is %d, should equal %d (number of observations)",
##                 length(offset), NROW(y)), domain = NA)
##     }
##     if (is.empty.model(mt)) {
##         x <- NULL
##         z <- list(coefficients = if (is.matrix(y)) matrix(NA_real_,
##             0, ncol(y)) else numeric(), residuals = y, fitted.values = 0 *
##             y, weights = w, rank = 0L, df.residual = if (!is.null(w)) sum(w !=
##             0) else if (is.matrix(y)) nrow(y) else length(y))
##         if (!is.null(offset)) {
##             z$fitted.values <- offset
##             z$residuals <- y - offset
##         }
##     }
##     else {
##         x <- model.matrix(mt, mf, contrasts)
##         z <- if (is.null(w))
##             lm.fit(x, y, offset = offset, singular.ok = singular.ok,
##                 ...)
##         else lm.wfit(x, y, w, offset = offset, singular.ok = singular.ok,
##             ...)
##     }
##     class(z) <- c(if (is.matrix(y)) "mlm", "lm")
##     z$na.action <- attr(mf, "na.action")
##     z$offset <- offset
##     z$contrasts <- attr(x, "contrasts")
##     z$xlevels <- .getXlevels(mt, mf)
##     z$call <- cl
##     z$terms <- mt
##     if (model)
##         z$model <- mf
##     if (ret.x)
##         z$x <- x
##     if (ret.y)
##         z$y <- y
##     if (!qr)
##         z$qr <- NULL
##     z
## }
## <bytecode: 0x564c78630a40>
## <environment: namespace:stats>
```

Don't feel overwhelmed by this. It is just to show you that built-in R functions are written in the same way as the R functions that you have written.

Iteration

The following shows how the integers from 1 to 8 can be summed using iteration via a `for` loop. Then this calculation is done in three other ways. It is usually best to avoid `for` loops if possible, as they are quite slow in R.

```
> res <- 0
> for (i in 1:8) res <- res + i
> res

## [1] 36

> sum(c(1, 2, 3, 4, 5, 6, 7, 8))

## [1] 36

> sum(1:8)
```

```
## [1] 36

> 8 *(8+1)/2

## [1] 36

> res <- 0
> for (i in 1:3) {
+         for (j in 1:4) {
+                 res <- res + i * j
+         }
+ }
> res     # Calculate the sum of i * j for all i = 1, 2, 3 and for all j = 1, 2, 3, 4.

## [1] 60
```

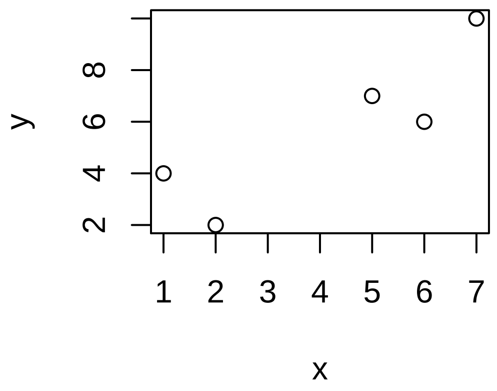
Random Numbers

R can generate (pseudo) random numbers in several different ways. Your random numbers will of course not be the same as these!

```
> help(runif)
> sampu <- runif(5)       # Generate 5 numbers randomly between 0 and 1
> sampu

## [1] 0.9538736 0.6468341 0.8640727 0.2778045 0.9577417

> mean(sampu)     # The average of the 5 random numbers

## [1] 0.7400653

> help(rnorm)
> sampn <- rnorm(100, 10, 2)      # Generate a random sample of 100 numbers
>                                 # from the normal distribution with
>                                 # mean 10 and standard deviation 2
> mean(sampn)

## [1] 10.03887

> quantile(sampn, 0.5)    # Find the median of the normal sample

##      50%
## 10.12197
```

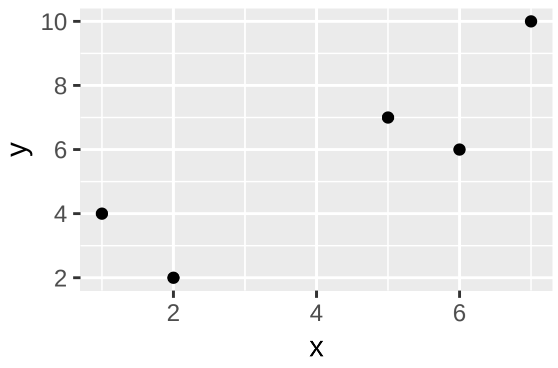
Graphics

```
> x <- c(1, 2, 5, 6, 7)
> y <- c(4, 2, 7, 6, 10)
> plot(x, y)      # Using "base" graphics
```

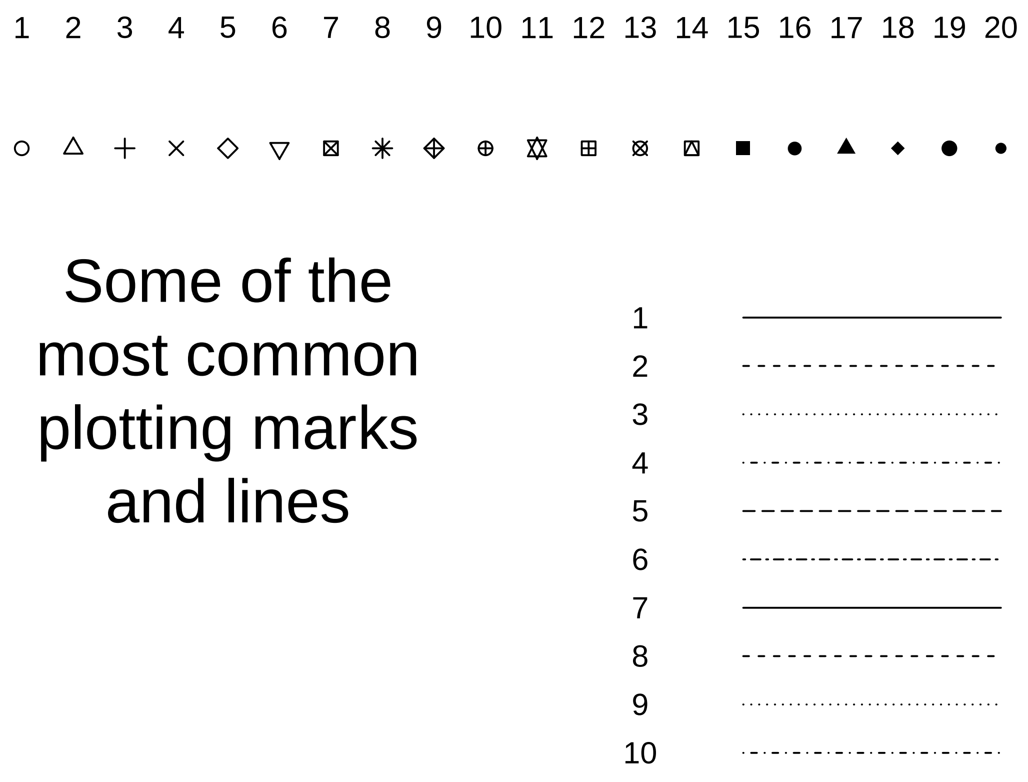


We will plot the same data using the ggplot2 package.

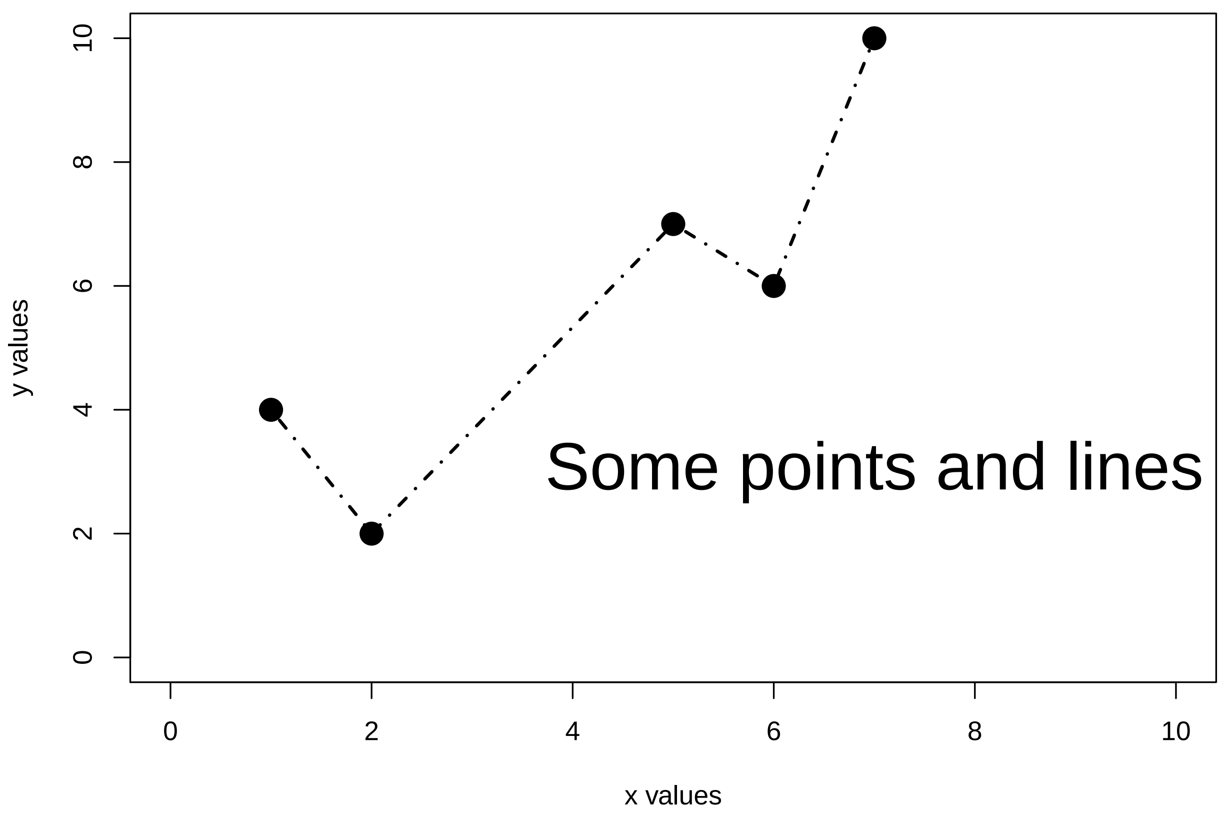
```
> # You may need to install ggplot2 using: install.packages("ggplot2")
> library(ggplot2)        # Using ggplot2 package

## Need help getting started?  Try the cookbook for R:
## http://www.cookbook-r.com/Graphs/

> dat <- data.frame(x = x, y = y)
> ggplot(aes(x = x, y = y), data = dat) + geom_point()
```



```
> help(par)        # Get information on plotting parameters for base graphics
> # Create a graph which displays some common plotting lines and marks:
> plot(c(1, 20), c(-0.33, 3.3), type = "n", xlab = "", ylab = "", axes = FALSE)
> for (i in 1:20) {text(i, 3, i); points(i, 2.5, pch = i)}
> for (i in 1:10) {text(13, 2 - 2 * i / 10, i); lines(c(15, 20), rep(2 - 2 * i / 10, 2), lty = i)}
> text(5, 1.5, "Some of the\nmost common\nplotting marks\nand lines", cex = 2)
```

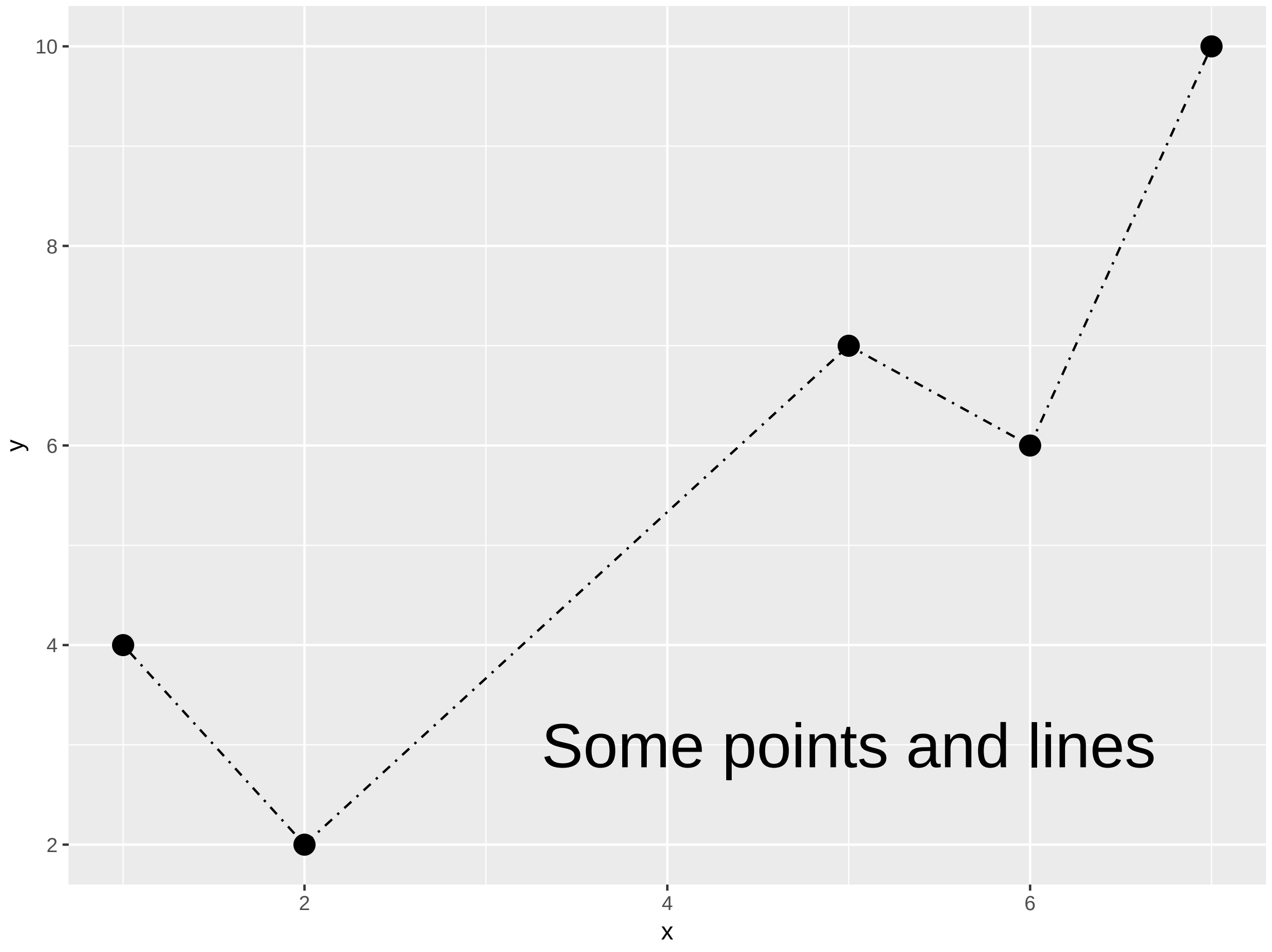


```
> plot(c(0, 10), c(0, 10), type = "n", xlab = "x values", ylab = "y values") # Create another graph
> points(x, y, pch = 16, cex = 2)
> lines(x, y, lty = 4, lwd = 2)
> text(7, 3, "Some points and lines", cex = 2.5)
```



Same but with `ggplot2`:

```
> ggplot(aes(x = x, y = y), data=dat) + geom_point(size = 4) + geom_line(linetype = 4) +
+          annotate("text", x = 5, y = 3, label = "Some points and lines", size = 10)
```

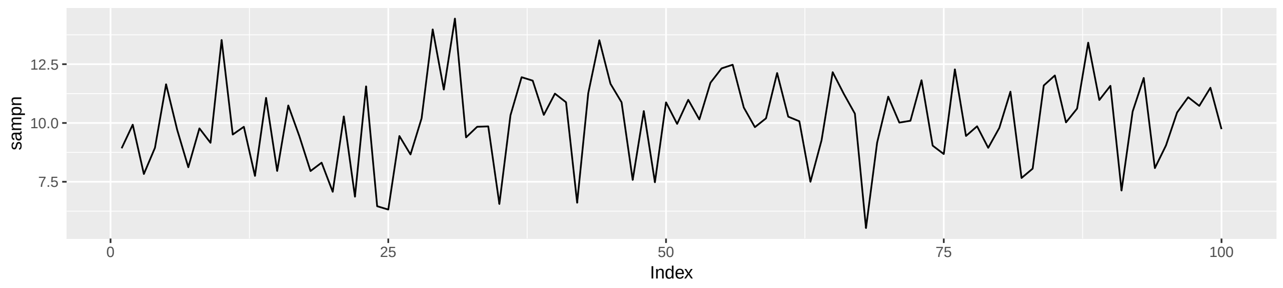


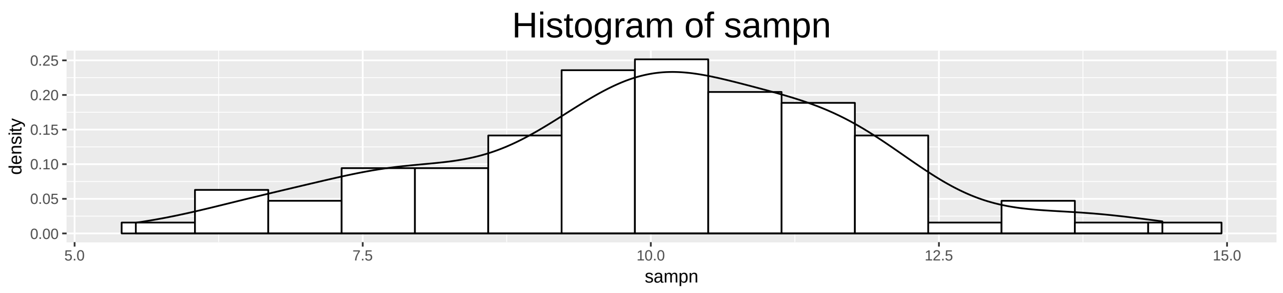
We can create PDF files which can be inserted into word processing documents.

```
> pdf("Graph.pdf")
> ggplot(aes(x = x, y = y), data=dat) + geom_point(size = 4) + geom_line(linetype = 4) +
+         annotate("text", x = 5, y = 3, label = "Some points and lines", size = 10)
> dev.off()

## pdf
##   2
```

Now find the file "Graph.pdf" in your working directory (use the command `getwd()` to find your working directory) and insert it into a Word document. You can also create graphs and save them as JPEG format for example. See `?Devices`. You can also save figures directly using RStudio.

```
> library(gridExtra)
> sampndat <- data.frame(sampn = sampn, Index = 1:length(sampn))
> p1 <- ggplot(aes(x = Index, y = sampn), data = sampndat) + geom_line()
> p2 <- ggplot(aes(x = sampn, y = ..density..), data = sampndat) +
+         geom_histogram(bins = 15, fill = "white", colour = "black") +
+         geom_density() + labs(title = "Histogram of sampn") +
+         theme(plot.title = element_text(hjust = 0.5, size = rel(2)))
> grid.arrange(p1, p2, nrow = 2)
```


sampn
12.5
10.0
7.5
0
25
50
75
100
Index


Histogram of sampn
density
0.25
0.20
0.15
0.10
0.05
0.00
5.0
7.5
10.0
12.5
15.0
sampn